SLB Assignment

Question-1: Task Master

- **By Adithya Ragothaman**

I have chosen to proceed with the Task Master Question.
I have created front end using HTML, CSS, and JavaScript, and the back end with a Flask framework using Python. For data storage, you can use SQLite database.

An overview of the files associated with the project.
(the bold letters are folders, the files are given below it)

1. **Static**
    a) Script.jss :
    The code enables users to manage tasks on a webpage, including adding, updating, and deleting tasks, with dynamic display and server interaction.
    b) Taskmasterstyles.css:
    The code provides CSS styling for a task management web page
2. **Templates**
    a) Loginlayout.html
    The HTML template provides a framework for a webpage, centered around Bootstrap.

    b) Loginpage.html
    Implements a login interface utilizing Bootstrap for styling, integrating form validation and Flask flash message handling for user input validation and feedback.

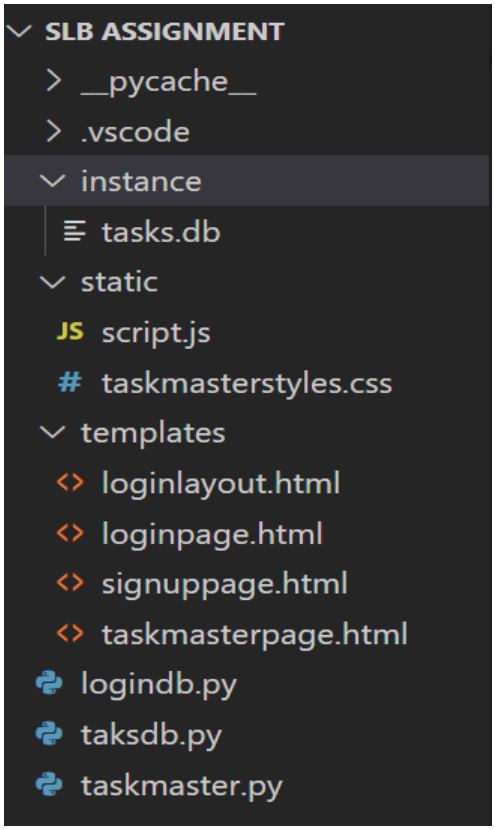
    c) Singuppage.html
    Features a Bootstrap-styled signup form with fields for username, email, and password, submitting data to "/signup.

    d) Taskmasterpage.html
    The HTML file creates a web page for a task management application with Bootstrap styling, a form for adding tasks, a table for displaying tasks, and JavaScript functions for updating and deleting tasks.
3. **Logindb.py**
    The script retrieves user details from a Flask application's database and prints them in tabular format.
4. **tasksdb.py**
    The script retrieves task details from a Flask application's database and prints them in a tabular format, including all the details of the tasks, using the tabulate library.

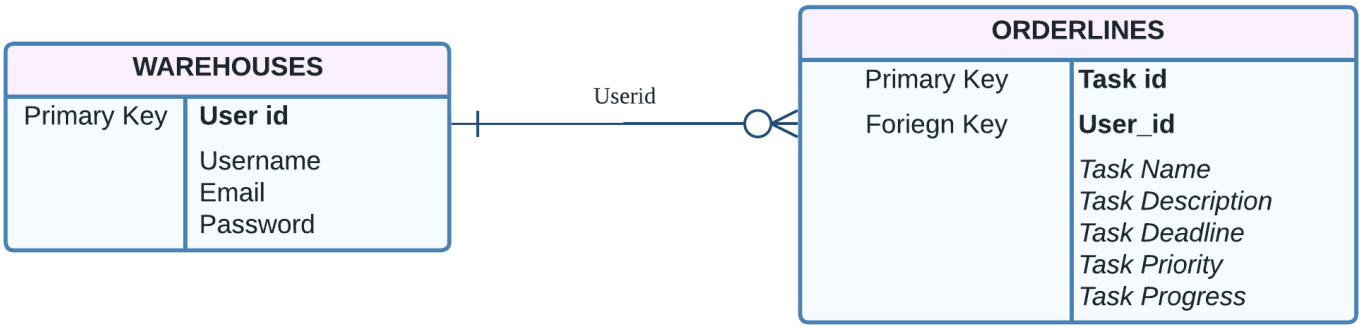
5. **taskmaster.py**

   This Python script creates a Flask web application for managing tasks with user authentication and database persistence. It allows users to sign up, log in, add, view, update, and delete tasks. The tasks are stored in an SQLite database, and the application provides RESTful APIs for interacting with tasks. Additionally, it includes features like password hashing, error handling, and session management.

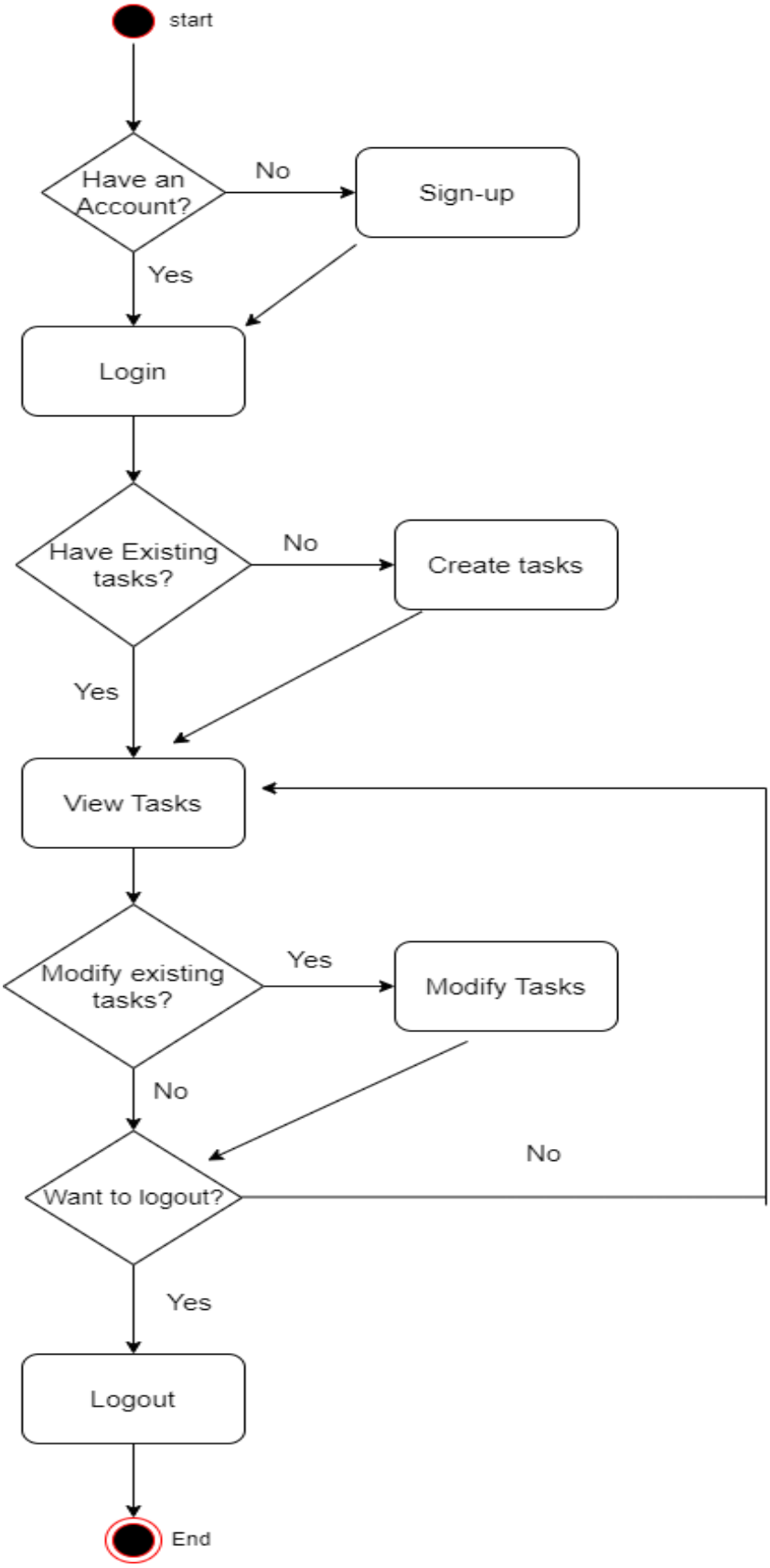
Run the taskmaster.py to run launch the application.



Database Design



Process Flow



The relationship between users and tasks is explicitly defined within the 'User' and 'Task' model classes. Here's how it's structured and what it implies about the relationship:

1. User Model:
   - Represents users of the application.
   - The 'User' class has a *'assigned_tasks'* relationship to the 'Task' class, indicating that each user can have multiple tasks assigned to them. This is a one-to-many relationship as defined by the 'db.relationship()' call, with backref='assigned_user' providing a way to access the user from a task, and 'lazy=True' indicating that the tasks are loaded on demand.
   - The 'primaryjoin' attribute in the relationship explicitly specifies the condition under which tasks are joined with users, reinforcing the linkage between the user's ID and the tasks 'user_id' field.

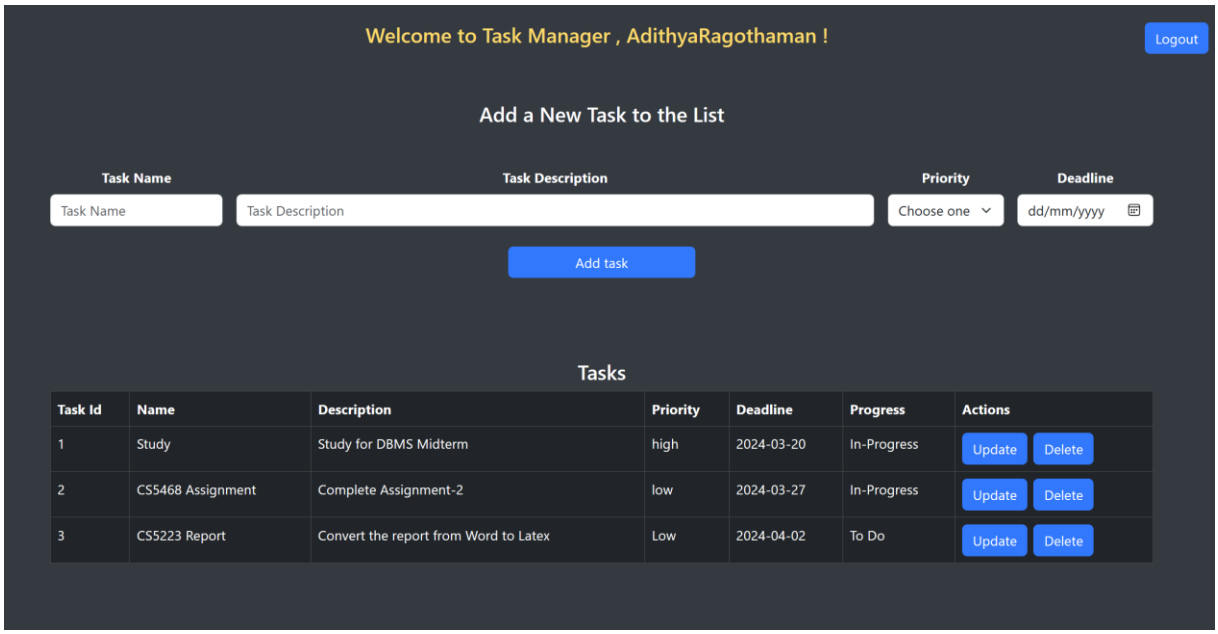
2. Task Model:
   - Represents tasks that can be created, viewed, updated, or deleted by authenticated users.
   - Each task has a '*user_id*' field, a foreign key linking the task to its owner (a user). This establishes the database-level relationship between tasks and users.
   - The 'user' relationship in the 'Task' class uses 'db.relationship()' to refer back to the 'User' model, allowing easy access from a task to its owner.

3. Application Routes:
   - The routes that handle task operations ('add_task', 'get_tasks', 'update_task', 'delete_task') enforce that tasks are managed in the context of the current authenticated user ('current_user' from Flask-Login). This ensures that users can only interact with their own tasks, further reinforcing the user-task relationship.

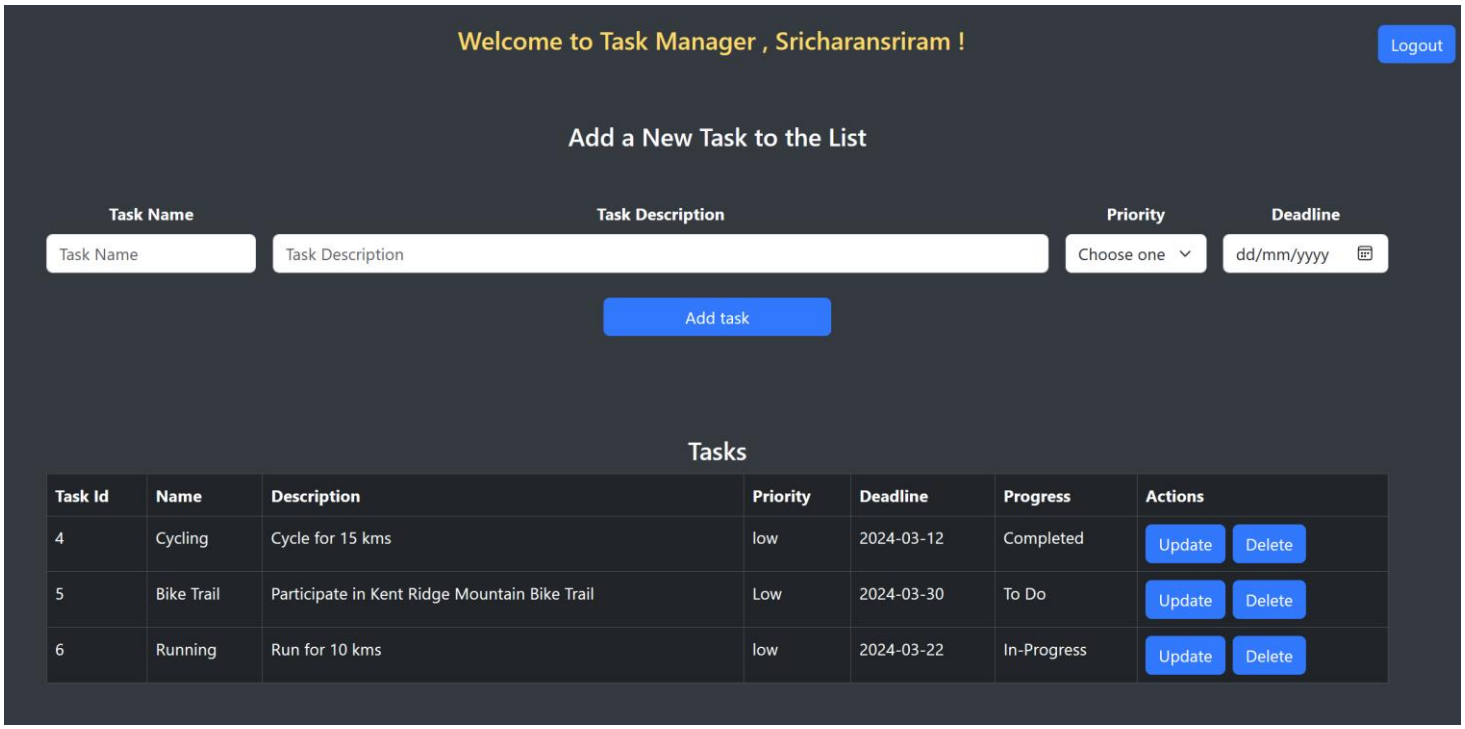
4. Authentication and User Session:
   - Flask-Login manages user sessions, ensuring that only authenticated users can access routes decorated with '@login_required'. The 'current_user' proxy is used to obtain the currently authenticated user, allowing the application to restrict task operations to tasks associated with 'current_user.id'.

5. Database Operations:
   - Tasks are filtered by 'user_id' when fetched, added, updated, or deleted, ensuring that users can only access and manipulate their tasks. This is a direct application of the one-to-many relationship between users and tasks in the database operations.

This code structure allows the application to maintain a secure, user-specific task management system, where the integrity and privacy of each user's tasks are preserved. The relationship is fundamental to the application's functionality, allowing it to offer personalized task management features.



Another user

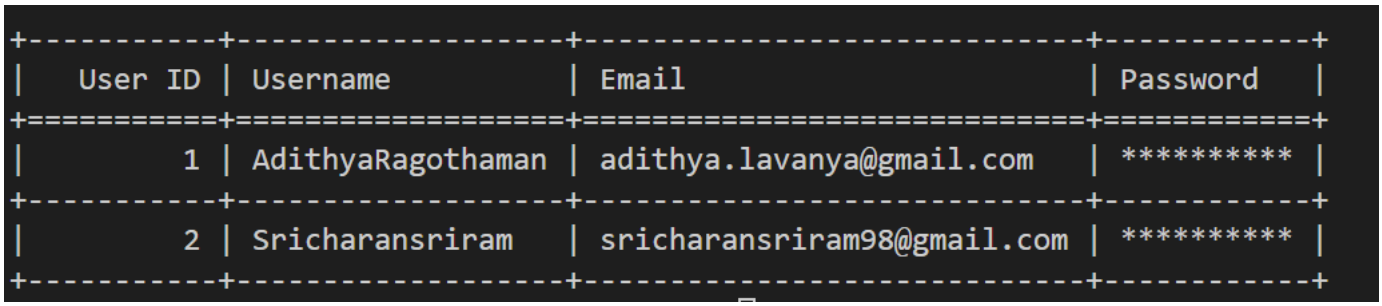


Look of tasks database

Task Id	Task Name	Task Description	Task Deadline	Task Priority	Task Progress	User ID
1	Study	Study for DBMS Midterm	2024-03-20	high	In-Progress	1
2	CS5468 Assignment	Complete Assignment-2	2024-03-27	low	In-Progress	1
3	CS5223 Report	Convert the report from Word to Latex	2024-04-02	Low	To Do	1
4	Cycling	Cycle for 15 kms	2024-03-12	low	Completed	2
5	Bike Trail	Participate in Kent Ridge Mountain Bike Trail	2024-03-30	Low	To Do	2
6	Running	Run for 10 kms	2024-03-22	low	In-Progress	2

The task id of users is displayed for users. The users need not worry about their task ids, it is for our reference in the database.

Look the userId is the last coloumn and so this table has both task and userid and can be queried.



User ID	Username	Email	Password
1	AdithyaRagothaman	adithya.lavanya@gmail.com	**********
2	Sricharansriram	sricharansriram98@gmail.com	**********

I have hashed and masked the password of the user.

These 2 db screenshots are from the output of the 2 python files I have attached

1. logindb.py

2. tasks.py

So 2 different databases are used for this project. (tasks and users)

Testing

To ensure that each endpoint and function behaves as expected in your Flask application, here are some unit tests you can implement, covering both positive and negative cases:

1. Signup Endpoint:

Positive Case: Test that a new user can sign up successfully with valid credentials. I Assert that the user is added to the database and receives a success message.

Negative Case: Attempt to sign up with an email that's already in use. I Assert that the operation is rejected and an appropriate error message is returned.

2. Login Endpoint:

Positive Case: Test that an existing user can log in with the correct credentials. I Assert that the login is successful and redirects to the index page.

Negative Case: Attempt to log in with incorrect credentials. I Assert that the login is denied and an error message is displayed.

3. Task Creation ('/tasks' POST):

Positive Case: Authenticated user submits a valid task. I Assert that the task is added to the database and a success response is returned.

Negative Case: Authenticated user submits a task with missing fields or a duplicate name. I Assert that the task is not added, and a validation error or conflict message is returned.

4. Fetch Tasks (‘/tasks’ GET):

Positive Case: Authenticated user requests their tasks. I Assert that all tasks associated with the user are returned successfully.

Negative Case: Unauthenticated user attempts to fetch tasks. I Assert that the request is rejected and redirects to the login page.

5. Update Task (‘/tasks’ PUT):

Positive Case: Authenticated user updates an existing task with valid changes. I Assert that the task is updated in the database and a success response is returned.

Negative Case: Authenticated user attempts to update a non-existing task or a task belonging to another user. I Assert that the update is rejected and an appropriate error message is returned.

6. Delete Task (‘/tasks ‘ DELETE):

Positive Case: Authenticated user deletes an existing task they own. I Assert that the task is removed from the database and a success response is returned.

Negative Case: Authenticated user attempts to delete a non-existing task or a task belonging to another user. I Assert that the deletion is rejected and an appropriate error message is returned.

Input Validation for LoginPage:

- Test that submitting the form with an empty email or password field prevents submission and prompts the user to fill in the required fields.
- Check that the email input field validates for a correctly formatted email address (using the pattern attribute).

Input Validation for SignupPage:

- Test that submitting the form with an empty email or password or username field prevents submission and prompts the user to fill in the required fields.
- Check that the email input field validates for a correctly formatted email address (using the pattern attribute).

Note-

* The application is dynamically built and wont require additional reloads.